**Objectives**

1. Implementation of the lexical analyzer using Flex

2. Identification and tokenization of various lexical elements, including keywords, identifiers, literals, and operators.

3. Implementation of grammar rules defined to capture the syntax of the programming language.

4. Describe the process of semantic analysis, checking for correctness in variable usage, type checking, and other semantic rules.

5. To use the gcc compiler and create a new programming language.

**Introduction**

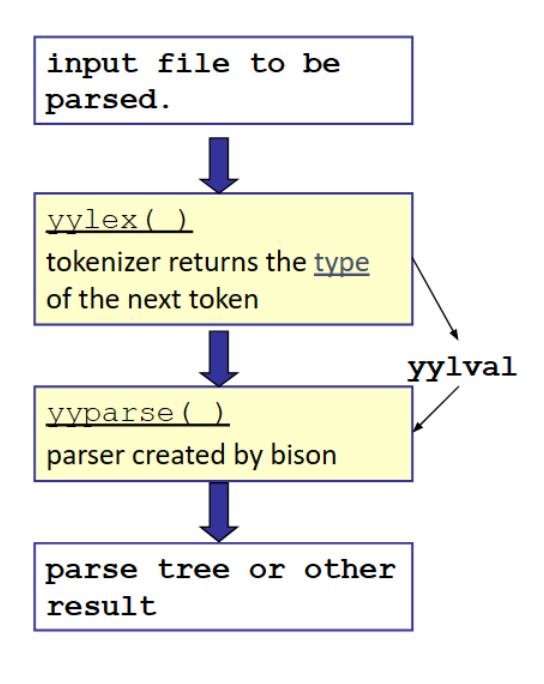
**FLEX :**

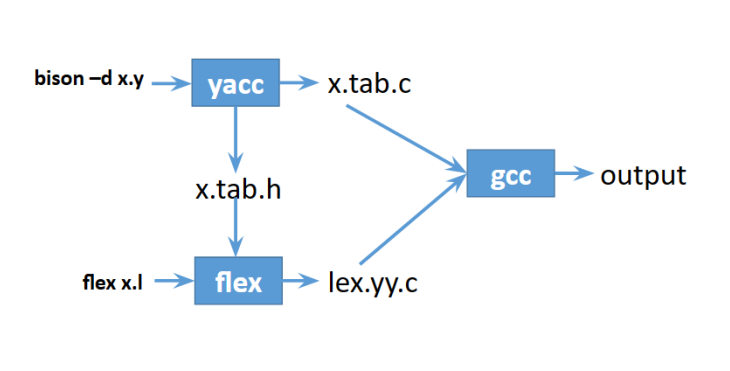
FLEX (Fast Lexical analyzer generator) is a tool for generating scanners. Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. Scanners perform lexical analysis by dividing the input into meaningful units. For a C program the units are variables, constants, keywords, operators, punctuation etc. These units also called as tokens.

**BISON :**

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Bison is used to perform semantic analysis in a compiler. Bison is a general-purpose parser generator that converts a grammar description for an LALR(1) context-free grammar into a C program to parse that grammar. Parsing involves finding the relationship between input tokens. Bison is upward compatible with Yacc: all properly-written Yacc grammars ought to work with Bison with no change. Interfaces with a scanner generated by Flex. Scanner is called as a subroutine when the parser needs the next token.

Flex and Bison are aging Unix utilities that help to write very fast parsers for almost arbitrary file formats. Flex and Bison will generate a parser that is virtually guaranteed to be faster than anything that could be written manually in a reasonable amount of time. Second, updating and fixing Flex and Bison source files is a lot easier than updating and fixing custom parser code. Third, Flex and Bison have mechanisms for error handling and recovery, finally Flex and Bison have been around for a long time, so they are far freer from bugs than newer code.





**Fig:** The parser process of a language and the union of flex and parse.

**Header File :**

AMDANI sample.h

AMDANI math.h

**Comment :**

NOTE This is a single-line comment

**Data Types :**

1. Integer:

Range: -2,147,483,648 to 2,147,483,647.

▪ PURNO: Returned if regular expression for detecting integers match an expression

▪ Syntax: PURNO ans; PURNO a = 5, b=9,c,d;

2. float:

Range: 1.2E-38 to 3.4E+38 Tokens:

▪ VOGNO: Returned if regular expression for detecting floating point numbers match

an expression

▪ Syntax: VOGNO a = 10.8;

3. String:

Range: a-z (small letters), A-Z (capital letters), 0-9 (digits) and symbols (:” ”)

▪ SOBDO: Returned if regular expression for detecting strings match an expression

▪ Syntax: SOBDO multiline = “This is a line string.”;

**Variable Declaration :**

STARTMAIN

PURNO a = 6;

PURNO b = 3;

PURNO summation = a JOG b ;

DEKHAW "Hello" JOG summation;

ENDMAIN

**Different ways of Variable Declaration with assignment :**

STARTMAIN

PURNO a = 5; // printf("Integer Variable, %s = %d\n", $3, varValue[$3]);

PURNO b = 5 , c,d,e;

PURNO upo = 5 JOG 6 , uno = b GUNN 8 , zuo = b BHAGGSHES a;

ENDMAIN

**Error in variable:**

cout << $1 << " is already declared!" << endl;

cout << $1 << " is not declared!" << endl;

**Input from User :**

"LIKHO" { return LIKHO; }

**Output Print function :**

"DEKHAW" { return DEKHAW; }

**Tokens:**

**Tokens used in this project:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Serial no.** | **Token** | **Input string** | **Realtime meaning of Token** |
| 1 | STARTMAIN | STARTMAIN | Start of program |
| 2 | ENDMAIN | ENDMAIN | End of program |
| 3 | STARTOFFUNCTION | STARTOFFUNCTION | Start of any function |
| 5 | ENDOFFUNCTION | ENDOFFUNCTION | End of function |
| 6 | HEADER | HEADER | Header Amdabu sample.h |
| 7 | JOG | JOG | + |
| 8 | BIYOG | BIYOG | - |
| 9 | GUNN | GUNN | \* |
| 10 | BHAGG | BHAGG | / |
| 11 | "BHAGGSHES" | "BHAGGSHES" | % |
| 12 | SUCHOK | SUCHOK | ^ |
| 13 | BARBEI | BARBEI | ++ |
| 14 | KOMBEI | KOMBEI | -- |
| 15 | POROMMAN | POROMMAN | |x| |
| 16 | RETURN | RETURN | Return of a declared function |
| 17 | PURNO | PURNO | Int |
| 18 | VOGNO | VOGNO | Float |
| 19 | SOBDO | SOBDO | String |
| 20 | SHARI | SHARI | Array |
| 21 | LAFAW | LAFAW | Switch |
| 22 | KOKHONO | KOKHONO | Case |
| 23 | VANGO | VANGO | Break |
| 24 | CHOLO | CHOLO | Continue |
| 25 | NORMALLY | NORMALLY | Default |
| 26 | NOT | NOT | ! |
| 27 | AND | AND | & |
| 28 | OR | OR | | |
| 29 | ANDAND | ANDAND | && |
| 30 | OROR | OROR | || |
| 31 | FACT | FACT | Factorial() |
| 32 | LEAPYEAR | LEAPYEAR | Leapyear function |
| 33 | SQRT | SQRT | Square root |
| 34 | GCD | GCD | Gcd store |
| 35 | LCM | LCM | Lcm function |
| 36 | MAX | MAX | Max function |
| 37 | MIN | MIN | Min function |
| 38 | PRIME | PRIME | Prime function |
| 39 | LESS | LESS | < |
| 40 | GREATER | GREATER | > |
| 41 | EQ | EQ | == |
| 42 | LEQ | LEQ | <= |
| 43 | GEQ | GEQ | >= |
| 44 | NEQ | NEQ | != |
| 45 | JOTOKHON | JOTOKHON | While |
| 46 | GHURO | GHURO | For |
| 47 | RANGE | RANGE | In |
| 48 | FROM | FROM | Range start |
| 49 | JODI | JODI | If |
| 50 | NAHOLE | NAHOLE | Else |
| 51 | JODINA | JODINA | Else if |
| 52 | TOBE | TOBE | Then (not user) |

Table 1. Used Tokens and their meanings

**Grammars Used in this Project:**

program : import STARTMAIN lines ENDMAIN;

import: HEADER

| /\*empty\*/

| HEADER import

| func

;

func: fun func

| fun

;

fun: STARTOFFUNCTION return\_type ID LFIRST arguments RFIRST lines ENDOFFUNCTION

;

return\_type:

| type

arguments:

| arguments arggg

;

arggg: type id2

| COMMA type id2

;

lines:

/\* empty \*/

| lines codes

;

declare:

type id SEMICOLON

;

type:

PURNO

| VOGNO

| SOBDO

;

id:

id2

| id COMMA id2

;

id2:

ID

| ID ASSIGN some

;

some:

some JOG some1

| some BIYOG some1

| some1

| FACT some

| LEAPYEAR some

| SQRT some

| GCD LFIRST some COMMA some RFIRST

| LCM LFIRST some COMMA some RFIRST

| MAX LFIRST some COMMA some RFIRST

| MIN LFIRST some COMMA some RFIRST

| PRIME LFIRST some RFIRST

;

some1:

some1 GUNN param\_last

| some1 BHAGG param\_last

| param\_last

;

param\_last :

arithmetic\_expression

| relational\_expression

| logical\_expression

| last\_id

;

arithmetic\_expression :

param\_last BHAGGSHES last\_id

| param\_last SUCHOK last\_id

;

relational\_expression :

param\_last LESS param\_last

| param\_last GREATER param\_last

| param\_last LEQ param\_last

| param\_last NEQ param\_last

| param\_last EQ param\_last

| param\_last GEQ param\_last

;

logical\_expression :

param\_last AND param\_last

| param\_last OR param\_last

| param\_last ANDAND param\_last

| param\_last OROR param\_last

| NOT param\_last

;

last\_id :

LFIRST some RFIRST

| WHOLENUMBER

| FRACTIONNUMBER

| STRINGLITERAL

| ID

;  
other:

other COMMA ID

| ID

;

codes:

SEMICOLON

| expression SEMICOLON

| declare

| DEKHAW putput SEMICOLON

| LIKHO type other SEMICOLON

| if\_blocksed

| for\_blocksed

| while\_blocksed

| do\_while\_blocksed

| switch\_blocksed

| RETURN last\_id SEMICOLON

;

putput:

| STRINGLITERAL putput

| STRINGLITERAL JOG showvariable putput

;

showvariable:

ID JOG

| ID

;

expression:

some

| ID ASSIGN ID LFIRST id RFIRST

| ID ASSIGN some

;

if\_blocksed:

JODI if\_block else\_statement

;

if\_block:

last\_id SEMICOLON LBRACE lines RBRACE

;

else\_statement:

| elif\_statement single\_else

| single\_else

;

single\_else:

NAHOLE LBRACE lines RBRACE

;

elif\_statement:

elif\_statement single\_elif

| single\_elif

;

single\_elif:

JODINA last\_id SEMICOLON LBRACE lines RBRACE

;.

for\_blocksed:

GHURO initialization SEMICOLON condional SEMICOLON incordec SEMICOLON LBRACE lines RBRACE

| GHURO RANGE some FROM some SEMICOLON LBRACE lines RBRACE

;

initialization:

| ID

| ID ASSIGN some

| initialization COMMA ID

| initialization COMMA ID ASSIGN some

;

condional:

| relational\_expression

;

incordec:

| ID ASSIGN some

| incordec COMMA ID ASSIGN some

;

while\_blocksed:

JOTOKHON condional SEMICOLON LBRACE lines RBRACE

;

do\_while\_blocksed:

OBOSSHOIKORO LBRACE lines RBRACE JOTOKHON condional SEMICOLON

;

switch\_blocksed:

LAFAW some SEMICOLON LBRACE sp\_code RBRACE

;

sp\_code: sp\_code1

| sp\_code1 NORMALLY SURU lines B

;

sp\_code1:

sp\_code1 KOKHONO some SURU lines B

| KOKHONO some SURU lines B

;

B:

VANGO SEMICOLON

| CHOLO SEMICOLON

;

**Discussion:**

The bison is used to parse a grammer. The assignment gives students a brief of how a parser work. The value stack in the parser is keeps the token value. The reduction of shift reduce problem and reduce/reduce problem was tried to remove but all weren’t possible to eliminate. With the help of respected teachers the concepts were cleared and so the assignment was possible to complete successfully. The input code is parsed using a bottom-up parser. As it is built with flex and bison, this compiler is unable to provide original functionality for if-else, loop, and switch case features. Some basic functionalities of a language were implemented. There are three basic data types that can be parsed using this compiler and they are Integer, Float and String. We can perform as basic arithmetic and logical operations. Some built-in functions such as sine, cosine, tangent, exponent, log etc. were also implemented. So, it can be said, the lab provides the students a wide knowledge of how a compiler works.

**Conclusion:**

The assignment was mainly to judge the students understanding on flex and bison. In order to build a language, flex and bison are very popular. It is the first step of lexical anlaysis and parsing. Designing a new language without a solid understanding of how a compiler works is a challenging task. In this project, a simple compiler was implemented using flex and bison. Several issues were encountered during the design phase of this compiler such as loop, if-else, functions etc. not working as they should owing to bison limitations. In the end, some of these issues were resolved. So overall the lab complete its objective of making students adapt with the huge knowledge of compiler process and all the task was able to solve individually.

**References:**

• Principles of Compiler Design By Alfred V.Aho & J.D Ullman

• www.geeksforgeeks/bison-for-parse